Avalanche: An Event Dispatcher for SNO+

(as in, a lot of sno(+), very fast)

Andy Mastbaum[*]

The University of Pennsylvania

May 24, 2012

Contents

1 Introduction **1**

2 Data Format **2**

3 Dispatched Data in RAT **2**

4 Dispatching Detector Events **2**

5 Writing Dispatcher Clients **3**

5.1 C++ 3

5.2 Python 4

5.3 Other Languages 4

6 Details **5**

7 Appendix: Packed Data Model **7**

1 Introduction

The "dispatcher" is the server software responsible for providing a stream of built events to various client software. Currently, client software includes the event viewer and detector monitoring tools. The dispatcher requires:

High Throughput The dispatcher must be able to keep up with the maximum data rate, simultaneously to all clients.

Scalability The dispatcher should support a large number of concurrent client connections without impact on performance.

Compatibility The dispatched data stream must be reasonably language-agnostic, as client software may be written in a variety of languages.

Relative to SNO, the SNO+ data stream will be split into two paths: event-level data which passes through the event builder, and run-level headers which are written to the database by the DAQ, manipulator control, and other systems. This means that monitoring applications may need to watch two sources: a "traditional" network dispatch from the builder, and the changes in a CouchDB database, merging them as shown in Figure 1.

[*]mastbaum@hep.upenn.edu

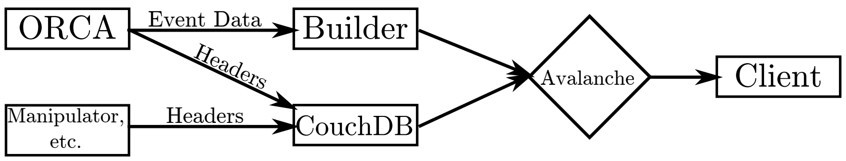

Figure 1: Data flow into avalanche clients

Python client and C++ server and client libraries to perform these tasks (excluding writing to CouchDB) are included in with avalanche in `lib`, and examples in both languages are given in the `examples/` directory.

2 Data Format

It has been decided that the "packed" events coming from the event builder will be in a ROOT format, with a structure developed by G. D. Orebi Gann and J. Kaspar. The packed format preserves the "raw" data structures from the detector front-end: for example, charge and time data is stored in 96-bit PMT bundles. This is in contrast the 'full' or 'unpacked' data structure used in RAT, where values are broken into separate, histogrammable fields.

The packed data is essentially a stream of `TObjects`, which may be event data, event-level headers, or run-level headers. For convenience of storage these are written to a `TTree`. Since `TTree`s are homogeneous lists, various data types must inherit from the same generic superclass to be stored in the tree. The data type must also be stored, so that objects can be properly re-cast when read back. The packed format `TTree` stores `RAT::DS::PackedRec` objects, which in turn contain a single `GenericRec` and an integer `RecordType`. All data types (events, headers) inherit from a class `GenericRec`, which is empty and exists only to support this polymorphism.

A full listing of packed format classes and their members is given in the appendix.

3 Dispatched Data in RAT

RAT includes an input producer and output processor for interaction with dispatched data. Monte Carlo events may be dispatched on the network, simulating the real data flow chain, and RAT can read in dispatched events, which may be used for online processing or writing monitoring tools as RAT processors.

The code to interact with dispatched data streams was committed to SVN in r652 (git revision f5387a0b495ca4b719a3a7e41487920b40f463bb). Documentation can be found in SNO+-doc-1300.

4 Dispatching Detector Events

A C++ library with a very simple API is provided in the `lib/cpp` directory. `libavalanche` provides both a dispatcher client `avalanche::client` and server `avalanche::server`. To build it, run `make`. This will create a shared library `libavalanche.so` which you may link into your application.

A server consists of a single `avalanche::server` object, constructed with a given socket address. The `avalanche::server` has one method, `sendObject`, which dispatches a ROOT `TObject`. For example, to send a histogram `TH1F* h1`:

```
avalanche::server* serv = new avalanche::server("tcp://localhost:5024");
if (serv->sendObject(h1) != 0)
  std::cout << "Couldn't send object" << std::endl;
```

Any number of clients may be subscribed to the stream on port 5024. `sendObject` returns nonzero if unsuccessful; checking the return value is recommended.

5 Writing Dispatcher Clients

Client software connects to a stream and/or database, receives and reconstructs ROOT objects, and does something with these objects. Libraries for C++ and Python are provided in `lib/` that manage the first two steps.

5.1 C++

`libavalanche`, located in `lib/cpp` provides both a dispatcher client `avalanche::client` and server `avalanche::server`. To build it, run `make`. This will create a shared library `libavalanche.so` which you may link into your application. You will also need to include the library header `avalanche.hpp` and, if connecting to CouchDB, `avalanche_rat.hpp`.

Consider the following example:

```
#include <iostream>
#include <TH1F.h>
#include <RAT/DS/PackedEvent.hh>
#include <avalanche.hpp>
#include <avalanche_rat.hpp>

int main(int argc, char* argv[]) {
    // create a client
    avalanche::client client;

    // connect to a few dispatchers
    client.addDispatcher("tcp://localhost:5025");
    client.addDispatcher("tcp://localhost:5024");

    // connect to couchdb
    avalanche::docObjectMap map = &(avalanche::docToRecord);
    client.addDB("http://username:password@localhost:5984", "db_name", map);

    // receive RAT::DS::PackedRec objects
    while (1) {
        RAT::DS::PackedRec* rec = (RAT::DS::PackedRec*) client.recv();
        if (rec) {
            std::cout << "Received PackedRec of type " << rec->RecordType << std::endl;
            if (rec->RecordType == 1) {
                RAT::DS::PackedEvent* event = dynamic_cast<RAT::DS::PackedEvent*> (rec->Rec);
                std::cout << " NHIT = " << event->NHits << std::endl;
            }
        }
        else
            continue;
        delete rec;
    }

    return 0;
}
```

First we create a client object, then we connect to a few dispatcher streams. You may connect to an unlimited number of streams; the packets received are interleaved, as if they arrived from one source. Next, we connect to a CouchDB server. Note the variable `map` – this tells avalanche how to turn CouchDB documents into ROOT `TObjects`. This function is provided for SNO+ use in `avalanche_rat.hpp` and can be used exactly as shown[1]. Finally, we receive ROOT objects with `recv`. Now we are ready to operate on the data. In this example we just print the record type and, if the record contains event data, the total NHIT.

Note that `recv` is non-blocking by default, and if no data is available will return NULL. To use blocking I/O, where `recv` will wait until data is received to return, use `recv(true)`.

5.2 Python

The `avalanche` Python package in `lib/python` provides a dispatcher client `avalanche.Client`. To install this package on your system, run `$ python setup.py install`.

Consider the following example:

```
import avalanche
import avalanche.ratdb
import sys
from rat import ROOT

if __name__ == '__main__':
    # create avalanche client
    client = avalanche.Client()

    # connect client to a dispatcher stream localhost port 5024
    client.add_dispatcher('tcp://localhost:5024')

    # connect client to a couchdb server at localhost:5984/dispatch
    doc_object_map = avalanche.ratdb.doc_to_record
    client.add_db('http://localhost:5984', 'changes_perf', doc_object_map,
                  username='username', password='password')

    # receive RAT::DS::PackedRec objects
    while True:
        rec = client.recv()
        if rec is not None:
            print 'Received PackedRec of type', rec.RecordType
```

First we create a client object, then we connect to a few dispatcher streams. You may connect to an unlimited number of streams; the packets received are interleaved, as if they arrived from one source. Next, we connect to a CouchDB server. Note the variable `doc_object_map` – this tells avalanche how to turn CouchDB documents into ROOT `TObjects`. This function is provided for SNO+ use in module `avalanche.ratdb` and can be used exactly as shown[2]. Finally, we receive ROOT objects with `recv`. Now we are ready to operate on the data. In this example we just print the record type a real client would at this point cast the member Rec object to the correct type, extract data, and perform some operations.

Note that `recv` is non-blocking by default, and if no data is available will return NULL. To use blocking I/O, where `recv` will wait until data is received to return, use `recv(blocking=True)`.

Note that in Python there is no need to cast the object returned by `recv`, as this is taken care of by PyROOT.

5.3 Other Languages

Many other languages are supported by ZeroMQ, and at least Ruby has supported ROOT bindings. Implementations will be similar to those given in C++ and Python. For examples of ZeroMQ in your language, see https://github.com/imatix/zguide/tree/master/examples.

[1]For more details, see `lib/cpp/README.md`.
[2]For more details, see `lib/python/README.md`.

As described above, a client must ‘subscribe’ to a ZeroMQ TCP ‘publish’ socket and deserialize ROOT `TObjects` using a `TBufferFile`. The former is achieved with the ZeroMQ API: set up a context and a subscriber socket, and connect that socket to one or more server addresses. The latter is simple in principle – create a `TBufferFile`, set the buffer to the packet contents, and read out the appropriate class. This may be complicated (as it is in Python) by the language’s hadling of string termination; care must be taken to ensure that the buffer is not treated as a string and truncated at the first null byte. For guidance, see the C++ and Python client library source code in `lib`.

CouchDB bindings also exist for many languages, but none are essential since Couch communicates using JSON strings sent over HTTP. To interact with a database, code needs to make HTTP queries and convert JSON strings into native objects. All major languages have such facilities. If no CouchDB library exists for your language, details on the CouchDB changes API are available at
http://wiki.apache.org/couchdb/HTTP_database _API#Changes.

6 Details

This section provides an overview or the implementations in the C++ and Python libraries, for those interested in modifying or extending them. Further details including complete API documentation can be found in `lib/cpp` (README and doxygen) and `lib/python` (README and sphinx).

6.1 Architecture

A ZeroMQ subscribe socket and a CouchDB changes feed are very different, so it is nontrivial to merge into one homogenous stream. This is accomplished using several threads which push received objects into a FIFO queue. The client’s `recv` method pops an element out of this queue (or `None`/`NULL` if it is empty). The blocking/non-blocking distinction doesn’t really propagate to the network level: the ZeroMQ listener is run in a non-blocking mode, and the CouchDB listener executes a callback.

ZeroMQ ROOT’s network classes (`TPSocket`, etc.) were considered for network I/O; however, the next-generation socket library ZeroMQ[3] provides a superior alternative. ZeroMQ is naturally asynchronous, natively supports fan-out, and implementations are typically faster than UNIX sockets. ZeroMQ’s fan-out allows clients to “subscribe” to a stream.

6.2 ROOT Dispatcher Stream

The `ROOT` native dispatcher stream is based on `TBufferFile`, `TMessage`, and `TSocket`. `TBufferFile` serializes a `TObject` into an array of characters using `TStreamerInfo` which is a part of a dictionary generated from a custom class. `TStremerInfo` is a recipe how to rearrange the class into a string and back, and is guaranteed to work across multiple architectures and `ROOT` versions. If you write a `ROOT` `TFile` it goes into the file header. `TMessage` is a light protocol wrapper around serialized `TObjects`. It can contain four different objects: serialized `TObject`, `StreamerInfo`, `ProcessIDs` (persistent links between different `TObjects`), and a string message. All these can be optionally compressed. Finally `TSocket` is a one-to-one socket guaranteeing that a client receives a `StreamerInfo` before the relevant `TObject`.

Our needs allow for a simplified desing. `ProcessIDs` and text strings are not supported. We want multiple clients to come and leave and to subscribe to a server which requires the clients to maintain their lists of `StreamInfos`. On the other hand a server needs to provide a tool for a client to request a missing `StreamerInfo`. Also we want all the clients to survive a server crash, e.i., servers come and leave, too.

Two channels are used between a server and a client. A `PUB`/`SUB` sockets handle `TObjects`, and `REP`/`REQ` sockets take care of `StreamInfos` distribution.A smart `ROUTER`/`DEALER` pair could do us a more efficient service, but a simple and easy to follow design was preferred.

The avalanche server broadcasts `TObjects` serialized using a ROOT `TBufferFile`. On the client side, software must create a `TBufferFile` using the packet data, read the `PackedRec` object out of it, and cast the `PackedRec`’s `Rec` object to the correct type based on the value of `RecordType`.

[3]http://www.zeromq.org

Since a ZeroMQ subscribe socket may connect to many publishers (servers), only one dispatcher-watching thread per client is needed.

6.3 CouchDB

Avalanche clients may also connect to a Couch database and receive headers as they are pushed in by the DAQ. These notifications are retrieved via the CouchDB changes feed[4]. Monitoring changes is done in Python using the `couchdb-python` package[5] and in C++ `libcurl` and `JSON-CPP`. Headers from CouchDB are converted into `TObject`s using a user-defined mapping function then pushed into the queue, and so interleaved with the rest of the data stream and indistinguishable from those received from an avalanche server. The mapping function takes a CouchDB document (dictionary-like in Python, a `Json::Value` in C++) and returns a `TObject*`.

CouchDB connections are independent, and each connection gets its own "watcher" thread.

[4]http://wiki.apache.org/couchdb/HTTP_database_API#Changes
[5]http://code.google.com/p/couchdb-python/

7 Appendix: Packed Data Model

This is a complete listing of all packed format classes, taken from RAT's PackedEvent.hh.

7.1 class GenericRec : public TObject

(empty)

7.2 class PackedRec : public TObject

- UInt_t RecordType
- GenericRec *Rec

7.2.1 Record Types

0. Empty
1. Detector event
2. RHDR
3. CAAC
4. CAST
5. TRIG
6. EPED

7.3 class PackedEvent : public GenericRec

- UInt_t MTCInfo[kNheaders] (6 words for the event header from the MTC)
- UInt_t RunID
- UInt_t SubRunID
- UInt_t NHits
- UInt_t EVOrder
- ULong64_t RunMask
- char PackVer
- char MCFlag
- char DataType
- char ClockStat10
- std::vector<PMTBundle> PMTBundles

7.4 class PMTBundle

- UInt_t Word[3]

7.5 class EPED : public GenericRec

- UInt_t GTDelayCoarse
- UInt_t GTDelayFine
- UInt_t QPedAmp
- UInt_t QPedWidth
- UInt_t PatternID
- UInt_t CalType
- UInt_t EventID (GTID of first events in this bank validity)
- UInt_t RunID (doublecheck on the run)

7.6 class TRIG : public GenericRec

- UInt_t TrigMask
- UShort_t Threshold[10]
- UShort_t TrigZeroOffset[10]
- UInt_t PulserRate
- UInt_t MTC_CSR
- UInt_t LockoutWidth
- UInt_t PrescaleFreq
- UInt_t EventID (GTID of first events in this banks validity)
- UInt_t RunID (doublecheck on the run)

7.6.1 Array Indices

Arrays correspond to:

0. N100Lo
1. N100Med
2. N100Hi
3. N20
4. N20LB
5. ESUMLo
6. ESUMHi
7. OWLn
8. OWLELo
9. OWLEHi

7.7 class RHDR : public GenericRec

- UInt_t Date
- UInt_t Time
- char DAQVer
- UInt_t CalibTrialID
- UInt_t SrcMask
- UInt_t RunMask
- UInt_t CrateMask
- UInt_t FirstEventID
- UInt_t ValidEventID
- UInt_t RunID (doublecheck on the run)

7.8 class CAST : public GenericRec

- UShort_t SourceID
- UShort_t SourceStat
- UShort_t NRopes
- float ManipPos[3]
- float ManipDest[3]
- float SrcPosUncert1
- float SrcPosUncert2[3]
- float LBallOrient
- std::vector<int> RopeID
- std::vector<float> RopeLen
- std::vector<float> RopeTargLen
- std::vector<float> RopeVel
- std::vector<float> RopeTens
- std::vector<float> RopeErr

7.9 class CAAC : public GenericRec

- float AVPos[3]
- float AVRoll[3] (roll, pitch and yaw)
- float AVRopeLength[7]